CS422 - Programming Language Design

From SOS to Rewriting Logic Definitions

Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign

In this chapter we show how SOS language definitions can be formalized using rewriting. More precisely, we show how one can *automatically* associate a rewrite logic theory in Maude to any SOS definition, be it big-step or small-step.

Rewrite Logic in Maude

Rewrite logic is an extension of equational logic with rewrite rules. We will discuss in more depth the distinction between equations and rewrite rules in rewriting logic in future lectures. For the time being, we need to make no operational distinction between them: they are both executed as rewrite rules, that is, from left to right.

As we saw in the previous lectures, in Maude equations are introduced using the keyword `eq`; for example:

```
eq term1 = term2 .
```

Rewrite rules are given using the keyword `rl` and replacing the equality symbol "`=`" by the rewrite symbol "`=>`":

```
rl term1 => term2 .
```

Conditional Equations and Rules

Both equations and rules can be conditional. In order to apply a conditional rules, one should first reduce the condition, also using term rewriting; the equation or rewrite rule is applied only if the condition is true. For example, the following is an equation (of rewrite rule) that is part of a bubble sort rewrite system:

```
ceq X,Y = Y,X if X > Y .
```

(or

```
crl X,Y => Y,X if X > Y .)
```

Note that "`eq`" and "`rl`" are replaced by "`ceq`" and "`rl`", respectively: "`c`" comes from "conditional".

There can be multiple conditions to equations or rules, and some conditions may involve matching or be themselves rules. We will discuss these on a "by need" basis.

Defining Syntax as an Algebraic Signature

As mentioned in previous lectures, the mix-fix notation for algebraic signatures is equivalent in expressivity to CFGs. Let us next see how we can formalize the syntax of our simple programming language as a signature in Maude. We next define a module without any equations or rules, its only role being to define the syntax of our language. Please compare the signature below to the CFG in the lectures on operational semantics.

We want our language to use Maude's *builtin libraries* of integers (as special arithmetic expressions) and identifiers (as variables), to avoid redefining them and thus focus on the more interesting aspects of our language definitions (nevertheless, one is free to define everything from scratch, including ones own version of integers and identifiers).

Therefore, we start by including the builtin modules `INT` and `QID`:

```
fmod SYNTAX is
  including INT .
  including QID .
```

The identifiers in `QID` are quoted: `'a`, `'abc`, `'a1`, `'x17`, etc. We next define the syntax of our variables: any quoted identifier is allowed to be a variable in our language, plus all the one letter "unquoted" identifiers; to achieve the latter, we define them as constants of sort `Var`:

```
--- Var
  sort Var .
  subsort Qid < Var .
  ops a b c d e f g h i j k l m n o p q r t u v x y z : -> Var .
```

We next define the syntax of arithmetic expressions. Recall that both variables and integers are arithmetic expressions. To avoid syntactic clashes with operations with the same name (addition, multiplication, etc.) already defined on integers, we prefer to "wrap" integers with an operator "`#`":

```
--- AExp
  sort AExp .
  subsort Var < AExp .
  op #_ : Int -> AExp [prec 30] .
  ops _+_ _-_ : AExp AExp -> AExp [prec 33 gather (E e)] .
  ops _*_ _/_ : AExp AExp -> AExp [prec 31 gather (E e)] .
```

Note that addition and subtraction have the same precedence, and also multiplication and division; all these operators are parsed as left associative, because of the attribute "`gather (E e)`".

Syntax for boolean expressions:

```
--- BExp
  sort BExp .
  ops true false : -> BExp .
  ops _<=_ _>=_ _==_ : AExp AExp -> BExp [prec 37] .
  op _and_ : BExp BExp -> BExp [prec 55] .
  op _or_ : BExp BExp -> BExp [prec 57] .
  op not_ : BExp -> BExp [prec 53] .
```

Syntax for statements:

```
--- Stmt
  sort Stmt .
  op skip : -> Stmt .
  op _:=_ : Var AExp -> Stmt [prec 40] .
  op _;_ : Stmt Stmt -> Stmt [gather (e E) prec 70] .
  op {_} : Stmt -> Stmt .
  op if_then_else_ : BExp Stmt Stmt -> Stmt [prec 60] .
  op while__ : BExp Stmt -> Stmt [prec 60] .

--- Pgm
  sort Pgm .
  op _;_ : Stmt AExp -> Pgm [prec 80] .
endfm
```

Parsing Programs

Now that the syntax is finished, we can write an parse programs. One may want to set the "print with parentheses" flag on, to better see the abstract syntax trees (AST); if one to see the ASTs in the more usual prefix form, then one can also set the mix-fix flag off:

```
set print with parentheses on .
parse x .

parse 'x + # 1 .

parse x - # 10 .

parse 'x := # 10 .

parse 'x := # 10 + 'x .
```

```
parse x + # 2 := # 10 .

parse x := # 1 ; y := x .

parse skip ; # 3 + y .

parse x := # 1 ; y := x ; y .

parse x := # 1 ; y + # 1 := x ; y .

parse
      x := # 1 ;
      y := (# 1 + x) * # 2 ;
      z := x * # 2 + x * y + y * # 2 ;
      x + y + z
 .
```

The following program calculates x^y:

```
parse
      x := # 17 ;
      y := # 100 ;
      p := # 1 ;
      i := y ;
      while not i == # 0 {
        p := p * x ;
        i := i - # 1
      } ;
      p
 .
```

The following program calculates the n^{th} Fibonacci's number:

```
parse
      x := # 0 ;
      y := # 1 ;
      n := # 1000 ;
      i := # 0 ;
      while not i == n {
        y := y + x ;
        x := y - x ;
        i := i + # 1
      } ;
      y
 .
```

The following program tests Collatz' conjecture for a given n:

```
parse
      n := # 178378342647823759743985734809582309829798347583490
      c := # 0 ;
      while not n == # 1 {
        c := c + # 1 ;
        if n == # 2 * (n / # 2)
        then n := n / # 2
        else n := # 3 * n + # 1
      } ;
      c
 .
```

We can now replace the keyword "`parse`" by "`rewrite`" and place all these programs in a file, say `sos-test.maude`, that we will use later to test our subsequent SOS definitions.

Defining the State

For our simple language, a state is nothing but a mapping of variables into values. This can be defined many different and equivalent ways, for example as a set of pairs "variable = integer".

We first define the constructors of such mappings:

```
fmod STATE is
  including SYNTAX .
  sort State .
  op _=_ : Var Int -> State .
  op __ : State State -> State [assoc comm id: empty ] .
  op empty : -> State .
```

Next we define the two operations on states, namely variable lookup and update:

```
  var S : State . var V : Var . var I I' : Int .

  --- retrieves a value from the state
  op _[_] : State Var -> Int .
  eq (S V = I)[V] = I .
  eq S[V] = 0 [owise] . --- default value 0

  --- updates a variable in the state
  op _[_<-_] : State Var Int -> State .
  eq (S V = I)[V <- I'] = S V = I' .
  eq S[V <- I] = S V = I [owise] .
endfm
```

Big-Step SOS in Rewriting Logic

Big step SOS can be defined in rewriting logic in a straightforward and automatic manner. Recall that in big-step SOS we had several types of configurations:

- $\langle a, \sigma \rangle$ and $\langle i \rangle$ where a is an arithmetic expression, σ a state and i an integer,
- $\langle b, \sigma \rangle$ and $\langle t \rangle$ where b is a boolean expression and t a truth value,
- $\langle s, \sigma \rangle$ and $\langle \sigma \rangle$ where s is a statement, and
- $\langle p \rangle$ where p is a program.

We need to define these explicitly:

```
fmod BIG-STEP-CONFIGURATION is
  including STATE .
  sort Configuration .
  op <_,_> : AExp State -> Configuration .
  op <_> : Int -> Configuration .
  op <_,_> : BExp State -> Configuration .
  op <_> : Bool -> Configuration .
  op <_,_> : Stmt State -> Configuration .
  op <_> : State -> Configuration .
  op <_> : Pgm -> Configuration .
endfm
```

We now define the SOS rules as conditional rewrite rules, one rewrite rule per SOS rule. The SOS rules and the corresponding rewriting logic rules are almost identical, the only difference being the slightly different syntax.

We prefer to group the rules for the different syntactic categories in modules. Each module includes the configurations, which implicitly includes the state, which includes the syntax:

```
mod AEXP-RULES is
  including BIG-STEP-CONFIGURATION .

  var X : Var .  var S : State .
  var I I1 I2 : Int . var A1 A2 : AExp .
```

We start with the trivial SOS rules for variable lookup and integer:

$$\frac{\cdot}{\langle x, \sigma \rangle \Downarrow \langle \sigma[x] \rangle} \quad (1)$$

$$\frac{\cdot}{\langle i, \sigma \rangle \Downarrow \langle i \rangle} \quad (2)$$

In rewriting logic, these rules become:

```
rl < X,S > => < S[X] > . ---var
rl < # I,S > => < I > .  ---int
```

Let us recall the big-step SOS rule for addition:

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle i_1\rangle,\ \langle a_2, \sigma\rangle \Downarrow \langle i_2\rangle}{\langle a_1 + a_2, \sigma\rangle \Downarrow \langle i\rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \quad (3)$$

In rewriting logic, it becomes:

```
crl < A1 + A2,S > => < I >
 if < A1,S > => < I1 >
 /\ < A2,S > => < I2 >
 /\ I := I1 + I2 .
```

Note that the side conditions become part of the condition of the rule now; indeed, the side conditions can be regarded as conditions under which corresponding rules apply. The only reason for which we preferred them as side conditions rather than hypotheses of rules when we defined SOS mathematically was because we did not want to include them as formal parts of proof objects.

However, when doing SOS derivations, one was still supposed to check the side conditions of SOS rules in order to see whether one applies correct instances of them, but the proof objects as such did not have to include an argument for why one was allowed to do it.

Since by defining them in rewriting logic we make the SOS rules *executable*, the side conditions need to also be checked somewhere. The simplest thing to do is to check them also as part of the condition. Note, however, that we use the matching operator ":=" for this purpose, not the rewrite relation "⇒"; this way, the side condition checking becomes a trivial and lighter-weight equational matching task, rather then a transition.

We can now similarly formalize all the remaining big-step SOS rules for arithmetic expressions and all those of boolean expressions:

```
  crl < A1 - A2,S > => < I >
   if < A1,S > => < I1 >
   /\ < A2,S > => < I2 >
   /\ I := I1 - I2 .

  crl < A1 * A2,S > => < I >
   if < A1,S > => < I1 >
   /\ < A2,S > => < I2 >
   /\ I := I1 * I2 .

  crl < A1 / A2,S > => < I >
   if < A1,S > => < I1 >
   /\ < A2,S > => < I2 >
   /\ not(I2 == 0)
   /\ I := I1 quo I2 .
endm
```

```
mod BEXP-RULES is
  including BIG-STEP-CONFIGURATION .
  var S : State .  var A1 A2 : AExp .  var B B1 B2 : BExp .
  var I I1 I2 : Int .  var T1 T2 : Bool .
  rl < true,S > => < true > .
  rl < false,S > => < false > .

  crl < A1 <= A2,S > => < true >
   if < A1,S > => < I1 >
   /\ < A2,S > => < I2 >
   /\ I1 <= I2 .

  crl < A1 <= A2,S > => < false >
   if < A1,S > => < I1 >
   /\ < A2,S > => < I2 >
   /\ I1 > I2 .
```

```
crl < A1 >= A2,S > => < true >
 if < A1,S > => < I1 >
 /\ < A2,S > => < I2 >
 /\ I1 >= I2 .


crl < A1 >= A2,S > => < false >
 if < A1,S > => < I1 >
 /\ < A2,S > => < I2 >
 /\ I1 < I2 .


crl < A1 == A2,S > => < true >
 if < A1,S > => < I >
 /\ < A2,S > => < I > .
```

```
crl < A1 == A2,S > => < false >
 if < A1,S > => < I1 >
 /\ < A2,S > => < I2 >
 /\ not(I1 == I2) .


crl < B1 and B2,S > => < true >
 if < B1,S > => < true >
 /\ < B2,S > => < true > .


crl < B1 and B2,S > => < false >
 if < B1,S > => < T1 >
 /\ < B2,S > => < T2 >
 /\ (T1 == false or T2 == false) .
```

```
  crl < B1 or B2,S > => < false >
   if < B1,S > => < false >
   /\ < B2,S > => < false > .

  crl < B1 or B2,S > => < true >
   if < B1,S > => < T1 >
   /\ < B2,S > => < T2 >
   /\ (T1 or T2) .

  crl < not B,S > => < true >
   if < B,S > => < false > .

  crl < not B,S > => < false >
   if < B,S > => < true > .
endm
```

```
mod STMT-RULES is
  including BIG-STEP-CONFIGURATION .

  var S S' S'' : State .  var St St1 St2 : Stmt .
  var A : AExp . var B : BExp . var X : Var . var I : Int .

  rl < skip,S > => < S > .

  crl < X := A,S > => < S[X <- I] >
   if < A,S > => < I > .

  crl < St1 ; St2,S > => < S' >
   if < St1,S > => < S'' >
   /\ < St2,S'' > => < S' > .
```

```
crl < {St},S > => < S' >
 if < St,S > => < S' > .


crl < if B then St1 else St2,S > => < S' >
 if < B,S > => < true >
 /\ < St1,S > => < S' > .


crl < if B then St1 else St2,S > => < S' >
 if < B,S > => < false >
 /\ < St2,S > => < S' > .


crl < while B St,S > => < S >
 if < B,S > => < false > .
```

```
  crl < while B St,S > => < S' >
   if < B,S > => < true >
   /\ < St,S > => < S'' >
   /\ < while B St,S'' > => < S' > .
endm
```

```
mod PGM-RULES is
  including BIG-STEP-CONFIGURATION .
  var St : Stmt . var A : AExp . var I : Int . var S : State .
  crl < St ; A > => < I >
   if < St,empty > => < S >
   /\ < A,S > => < I > .
endm

mod BIGSTEP-SEMANTICS is
  including AEXP-RULES .
  including BEXP-RULES .
  including STMT-RULES .
  including PGM-RULES .
endm
```

Small-Step SOS in Rewriting Logic

We now show how a small-step SOS semantics can also be formalized into rewriting logic. The procedure is a bit more involved than the one for big-step semantics, but it is still *entirely automatic* and *one-to-one faithful* to the original small-step semantics.

The reason for which the big-step SOS was so easy to formalize in rewriting logic was that rewriting logic conceptually is itself a "big-step" semantics: the rewrite relation "$\Rightarrow$" in Maude is a transitive closure of the smaller rewrite steps.

More precisely, a conditional rule:

$$\mathsf{crl}\ t \Rightarrow t'\ \mathsf{if}\ u \Rightarrow u'$$

is applied as follows:

- match t against the term to rewrite, obtaining a substitution θ
- check whether $\theta(u)$ rewrites to $\theta(u')$ in *zero*, *one* or *more steps*. u' can have more variables than u; if this is the case, then a larger substitution than θ is obtained, say θ'
- conclude that $\theta(t) \rightarrow \theta'(t')$ in *one step*.

Recall the big-step semantics of addition in rewriting logic:

```
crl < A1 + A2,S > => < I >
 if < A1,S > => < I1 >
 /\ < A2,S > => < I2 >
 /\ I := I1 + I2 .
```

The rewrites in the condition are therefore reflexive and transitive closures of small steps. Fortunately, because of how the big-step semantics is formalized, there is only one way that each of the conditions holds in zero, one or more steps: the expressions are completely evaluated to `I1` and `I2`, respectively. This happens to be exactly what we wanted in the case of big-step SOS.

Suppose now that one would attempt to formalize the small-step SOS semantics by simply replacing the arrow in SOS with the arrow in rewriting logic. For example, the small-step SOS semantics of addition, which is

$$\frac{\langle a_1, \sigma\rangle \rightarrow \langle a_1', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1' + a_2, \sigma\rangle} \tag{4}$$

$$\frac{\langle a_2, \sigma\rangle \rightarrow \langle a_2', \sigma\rangle}{\langle a_1 + a_2, \sigma\rangle \rightarrow \langle a_1 + a_2', \sigma\rangle} \tag{5}$$

$$\frac{\cdot}{\langle i_1 + i_2, \sigma\rangle \rightarrow \langle i, \sigma\rangle} \text{ where } i \text{ is the sum of } i_1 \text{ and } i_2 \tag{6}$$

would be:

```
crl < A1 + A2,S > => < A1' + A2,S >
 if < A1,S > => < A1',S > .

crl < A1 + A2,S > => < A1 + A2',S >
 if < A2,S > => < A2',S > .

crl < # I1 + # I2,S > => < # I,S >
 if I := I1 + I2 .
```

While this may look right at first sight and may even have a chance to work when executed (depending upon how many steps the underlying engine choose to try in the condition before declaring it "satisfied" and returning the corresponding substitution), it is in fact *not* semantically faithful to small-step SOS.

That is because there is nothing to enforce *only one step* reductions in the condition rewrites of the conditional rules, as it happens in small-step SOS.

For example, the condition "`< A1,S > => < A1',S >`" can succeed in zero, one or more steps in rewriting logic, but what we want in small-step SOS is to have it succeed in precisely one step. In particular, a rewrite engine like Maude has it succeed in zero steps, in which case `A1'` is nothing but `A1` and therefore loops forever.

What one would like is to write the same rules as above, but to somehow replace the many-step => by a one step variant, say =>1,

```
crl < A1 + A2,S > =>1 < A1' + A2,S >
 if < A1,S > =>1 < A1',S > .

crl < A1 + A2,S > =>1 < A1 + A2',S >
 if < A2,S > =>1 < A2',S > .

crl < # I1 + # I2,S > =>1 < # I,S >
 if I := I1 + I2 .
```

and then to define => as the reflexive and transitive closure of =>1.

Supposing that one somehow defines the relation `=>1`, then it would be very easy to define `=>`:

```
crl < A,S > => < A',S >  if < A,S > =>1 < A',S > .
```

A rewrite command "`rewrite < A,S >`" would now iteratively apply the conditional rule above calling the one step relation iteratively, thus obtaining a faithful, one-to-one formalization of small-step SOS into rewriting logic.

Unfortunately, there is apparently no way to define `=>1` ...

Fortunately, since SOS rules apply only at the top of the configurations, there is a simple *structural trick* that we can use to *inhibit* the rewrite relation and make it apply only one step in conditions, and thus to simulate the relation `=>1`.

The idea is to introduce *two new types of configurations*:

- We introduce configurations wrapped by curly brackets, of the form { `A,S` }, with the intuitive meaning that these are *fresh configurations* on which a one step rewrite step can be applied.
- We also introduce configurations wrapped by square brackets, of the form `[ A,S ]`, with the intuition that these are *consumed configurations* on which no step can be applied.

With these, we can simulate a one-step rewrite as follows:

```
crl { A1 + A2,S } => [ A1' + A2,S ]
 if { A1,S } => [ A1',S ] .

crl { A1 + A2,S } => [ A1 + A2',S ]
 if { A2,S } => [ A2',S ] .

crl { # I1 + # I2,S } => [ # I,S ]
 if I := I1 + I2 .
```

Once such a simulated one-step relation is defined, one can define its transitive closure on actual configurations as follows:

```
crl < A,S > => < A',S >
 if { A,S } => [ A',S ] .
```

Let us now formalize this technique. We start by defining all the configurations, which are, as seen above, three different ones for each configuration in the small-step SOS definition.

```
fmod SMALL-STEP-CONFIGURATION is
  including STATE .
  sort Configuration .
```

We first add the configurations for arithmetic expressions:

```
  op <_,_> : AExp State -> Configuration .
  op {_,_} : AExp State -> Configuration .
  op [_,_] : AExp State -> Configuration .
```

Then those for boolean expressions:

```
  op <_,_> : BExp State -> Configuration .
  op {_,_} : BExp State -> Configuration .
  op [_,_] : BExp State -> Configuration .
```

And then those for statements:

```
op <_,_> : Stmt State -> Configuration .
op {_,_} : Stmt State -> Configuration .
op [_,_] : Stmt State -> Configuration .
```

Recall that our small-step SOS also had configurations which were just plain states; these were needed to define the semantics of statements that "dissolved", such as

$$\frac{\cdot}{\langle \texttt{skip}, \sigma \rangle \rightarrow \langle \sigma \rangle} \tag{7}$$

$$\frac{\cdot}{\langle x := i, \sigma \rangle \rightarrow \langle \sigma[x \leftarrow i] \rangle} \tag{8}$$

The plain-state configurations are never transformed, they are only the results of other transformations. Hence, we do not need a fresh configuration variant for the plain-state configurations:

```
op <_> : State -> Configuration .
op [_] : State -> Configuration .
```

In our small-step SOS we also had (initial) configurations that held only a program; such configurations were never generated by rules, so we do not need to define a "consumed" configuration variant:

The next two configurations are

```
  op <_> : Pgm -> Configuration .
  op {_} : Pgm -> Configuration .
```

We also had configurations of programs and states:

```
  op <_,_> : Pgm State -> Configuration .
  op {_,_} : Pgm State -> Configuration .
  op [_,_] : Pgm State -> Configuration .
```

The final result of the evaluation of the program, an integer, was also a configuration:

```
  op <_> : Int -> Configuration .
endfm
```

It is now a simple exercise to formalize all the small-step SOS rules for expressions, statements and programs as rewrite rules simulating the one-step rewrite using the auxiliary configurations. What is a bit trickier is to calculate the complete transitive closure of the one step relation.

```
mod AEXP-RULES is
  including SMALL-STEP-CONFIGURATION .
  var X : Var .  var S : State .
  var I I1 I2 : Int . var A1 A2 A1' A2' : AExp .

  rl {X,S} => [# (S[X]),S] . ---var

  crl {A1 + A2,S} => [A1' + A2,S]
   if {A1,S} => [A1',S] .

  crl {A1 + A2,S} => [A1 + A2',S]
   if {A2,S} => [A2',S] .

  crl {# I1 + # I2,S} => [# I,S]
   if I := I1 + I2 .
```

```
crl {A1 - A2,S} => [A1' - A2,S]
 if {A1,S} => [A1',S] .

crl {A1 - A2,S} => [A1 - A2',S]
 if {A2,S} => [A2',S] .

crl {# I1 - # I2,S} => [# I,S]
 if I := I1 - I2 .

crl {A1 * A2,S} => [A1' * A2,S]
 if {A1,S} => [A1',S] .

crl {A1 * A2,S} => [A1 * A2',S]
 if {A2,S} => [A2',S] .
```

```
  crl {# I1 * # I2,S} => [# I,S]
   if I := I1 * I2 .

  crl {A1 / A2,S} => [A1' / A2,S]
   if {A1,S} => [A1',S] .

  crl {A1 / A2,S} => [A1 / A2',S]
   if {A2,S} => [A2',S] .

  crl {# I1 / # I2,S} => [# I,S]
   if not(I2 == 0)
   /\ I := I1 quo I2 .
endm
```

```
mod BEXP-RULES is
  including SMALL-STEP-CONFIGURATION .

  var S : State .  var A1 A2 A1' A2' : AExp .
  var B B1 B2 B' B1' B2' : BExp . var I I1 I2 : Int .

  crl {A1 <= A2,S} => [A1' <= A2,S]
   if {A1,S} => [A1',S] .

  crl {A1 <= A2,S} => [A1 <= A2',S]
   if {A2,S} => [A2',S] .

  crl {# I1 <= # I2,S} => [true,S]
   if I1 <= I2 .
```

```
crl {# I1 <= # I2,S} => [false,S]
 if I1 > I2 .

crl {A1 >= A2,S} => [A1' >= A2,S]
 if {A1,S} => [A1',S] .

crl {A1 >= A2,S} => [A1 >= A2',S]
 if {A2,S} => [A2',S] .

crl {# I1 >= # I2,S} => [true,S]
 if I1 >= I2 .

crl {# I1 >= # I2,S} => [false,S]
 if I1 < I2 .
```

```
crl {A1 == A2,S} => [A1' == A2,S]
 if {A1,S} => [A1',S] .

crl {A1 == A2,S} => [A1 == A2',S]
 if {A2,S} => [A2',S] .

rl {# I == # I,S} => [true,S] .

crl {# I1 == # I2,S} => [false,S]
 if not(I1 == I2) .

crl {B1 and B2,S} => [B1' and B2,S]
 if {B1,S} => [B1',S] .

crl {B1 and B2,S} => [B1 and B2',S]
 if {B2,S} => [B2',S] .
```

```
rl {true and true,S} => [true,S] .
rl {true and false,S} => [false,S] .
rl {false and true,S} => [false,S] .
rl {false and false,S} => [false,S] .

crl {B1 or B2,S} => [B1' or B2,S]
 if {B1,S} => [B1',S] .

crl {B1 or B2,S} => [B1 or B2',S]
 if {B2,S} => [B2',S] .

rl {true or true,S} => [true,S] .
rl {true or false,S} => [true,S] .
rl {false or true,S} => [true,S] .
rl {false and false,S} => [false,S] .
```

```
  crl {not B,S} => [not B',S]
   if {B,S} => [B',S] .

  rl {not true,S} => [false,S] .
  rl {not false,S} => [true,S] .
endm
```

```
mod STMT-RULES is
  including SMALL-STEP-CONFIGURATION .

  var S S' S'' : State .  var St St1 St2 St' St1' : Stmt .
  var A A' : AExp . var B B' : BExp .  var X : Var . var I : Int

  rl {skip,S} => [S] .

  crl {X := A,S} => [X := A',S]
   if {A,S} => [A',S] .

  rl {X := # I,S} => [S[X <- I]] .

  crl {St1 ; St2,S} => [St1' ; St2,S']
   if {St1,S} => [St1',S'] .
```

```
crl {St1 ; St2,S} => [St2,S']
 if {St1,S} => [S'] .

crl {{St},S} => [{St'},S']
 if {St,S} => [St',S'] .

crl {{St},S} => [S']
 if {St,S} => [S'] .

crl {if B then St1 else St2,S} => [if B' then St1 else St2,S]
 if {B,S} => [B',S] .

rl {if true then St1 else St2,S} => [St1,S] .
rl {if false then St1 else St2,S} => [St2,S] .

rl {while B St,S} => [if B then (St ; while B St) else skip,S]
```

```
endm

mod PGM-RULES is
  including SMALL-STEP-CONFIGURATION .
  var St St' : Stmt . var A : AExp . var S S' : State . var P : P
  rl {P} => [P,empty] .

  crl {St ; A,S} => [St' ; A,S']
   if {St,S} => [St',S'] .

  crl {St ; A,S} => [A,S']
   if {St,S} => [S'] .
endm
```

The only way to execute programs or fragments of programs under a small-step SOS of a language, is to compute the transitive closure of the small-step relation. We do it in what follows:

```
mod SMALLSTEP-SEMANTICS is
  including AEXP-RULES .
  including BEXP-RULES .
  including STMT-RULES .
  including PGM-RULES .

  var A A' : AExp .  var S S' : State .  var I : Int .
  var B B' : BExp . var St St' : Stmt . var P P' : Pgm .
```

Let us first define the transitive closures of the trivial one-step relations:

```
crl < A,S > => < A',S' >
 if {A,S} => [A',S'] .

crl < B,S > => < B',S' >
 if {B,S} => [B',S'] .

crl < St,S > => < St',S' >
 if {St,S} => [St',S'] .

crl < P,S > => < P',S' >
 if {P,S} => [P',S']
.
```

Recall that a program could either advance its statement by one step, or, if that dissolved, move on to the evaluation of its terminal arithmetic expression. We treated the first case above; for the second, we add the following rule establishing the transition from a program/state configuration to an expression/state one:

```
crl < P,S > => < A,S' >
 if {P,S} => [A,S'] .
```

If a statement dissolved, then it transited to a "consumed" state; for uniformity and for the sake of mechanization of our translation from SOS to rewriting logic, we introduced an angle-bracket configuration notation for state-configurations even though, strictly speaking, that was not necessary. We also add a corresponding rule for moving the statement-dissolve step up at the top level:

```
crl < St,S > => < S' >
 if {St,S} => [S'] .
```

We also want to capture the initiation of the reduction as a top-level step:

```
  crl < P > => < P,S >
   if {P} => [P,S] .
```

and so we want for the termination of the computation:

```
  rl < # I,S > => < I > .
endm
```